

Macros for User-Friendly Systems

Bob Virgile, Robert Virgile Associates, Inc.

Overview

Macro-based systems can make life easy for the end user. However, system development requires a programmer with good SAS(R) skills as well as macro skills. This paper explores some of the techniques for displaying data, responding to user selections, and minimizing wear and tear on the user.

The programmer's SAS tools should include extensive familiarity with the manipulation and transformation of SAS datasets. For example, this program should be 100% clear to any programmer developing macro-based systems:

```
DATA POOL1 POOL2 POOL3 POOL4;
SET ALLPOOLS;
IF POOL=1 THEN OUTPUT POOL1;
ELSE IF POOL=2 THEN OUTPUT POOL2;
ELSE IF POOL=3 THEN OUTPUT POOL3;
ELSE IF POOL=4 THEN OUTPUT POOL4;
```

Equally important, the macro code which generates these statements must be 100% clear. Given that &NPOOLS resolves to 4, the macro code might be:

```
DATA %DO DUMMY=1 %TO &NPOOLS;
    POOL&DUMMY
%END;
;
SET ALLPOOLS;
%DO DUMMY=1 %TO &NPOOLS;
    %IF &DUMMY>1 %THEN ELSE;
    IF POOL=&DUMMY THEN
        OUTPUT POOL&DUMMY;
%END;
```

Since (1) each macro statement ends in a semicolon, and (2) a macro statement may generate a portion of a SAS language statement, many semicolons can appear before reaching the end of a SAS statement. Of course, the entire DATA step must be embedded within a macro definition, since %DO is not permitted in open SAS code.

Details

Macro-based systems can employ dozens of techniques for manipulating and displaying data. This paper will cover only a small sample of techniques. The examples use DATA step windows to display data, because windows can be defined on the fly using macro language.

The best structure for storing data often differs from the best form for collecting or displaying data. A programmer must be adept at transforming data from one form to another. If the data were to contain 9 observations:

NAME	PLACE	COST
ALICE	ARRUBA	20
ALICE	ARRUBA	30
BOB	ARRUBA	20
CAROL	ARRUBA	35
TED	ARRUBA	29
ALICE	BERMUDA	15
BOB	BERMUDA	40
CAROL	BERMUDA	18
TED	BERMUDA	27

A reasonable display might contain one observation and 14 variables:

	ARRUBA	BERMUDA
ALICE	50	15
BOB	20	40
CAROL	35	18
TED	29	27

Data transformation can involve summarizing, transposing, or otherwise re-shaping data for display. The example above might require a two-step process, beginning with a summary:

```

PROC SUMMARY DATA=VACATION NWAY;
CLASS PLACE NAME;
VAR COST;
OUTPUT OUT=TOTALS
(DROP=_FREQ_ _TYPE_)
SUM=TOTCOST;

```

If we knew there would be two PLACES and four NAMES, the next program would reorganize the TOTCOST variable into one observation.

```

DATA ONEOBS;
ARRAY COSTS {8} ROW1COL1-ROW1COL2
                ROW2COL1-ROW2COL2
                ROW3COL1-ROW3COL2
                ROW4COL1-ROW4COL2;
DO DUMMY=1 TO 8;
    SET TOTALS;
    COSTS{DUMMY} = TOTCOST;
END;
OUTPUT;
STOP;

```

Now let's add a few of the missing pieces. The NAMES and PLACES need to be incorporated, and the actual number of each may be unknown. We will assume some number of NAMES from 1 to 9, and some number of PLACES from 1 to 5. The program must count the number of each, and set up corresponding variables to hold COST, NAME, and PLACE information. Begin by summarizing a summary dataset:

```

PROC SUMMARY DATA=TOTALS NWAY;
CLASS PLACE;
OUTPUT OUT=PLACES (KEEP=PLACE);

```

```

PROC SUMMARY DATA=TOTALS NWAY;
CLASS NAME;
OUTPUT OUT=NAMES (KEEP=NAME);

```

```

DATA _NULL_;
IF 0 THEN DO;
    SET NAMES POINT=_N_ NOBS=NROWS;
    SET PLACES POINT=_N_ NOBS=NCOLS;
END;
CALL SYMPUT('NROWS', PUT(NROWS, 1.));
CALL SYMPUT('NCOLS', PUT(NCOLS, 1.));
STOP;

```

A few notes here. The STOP statement is absolutely necessary to prevent the DATA step from looping. Most of the time when POINT is used, a STOP statement will be necessary for just that purpose. Secondly, many operating systems allow NOBS= without using POINT=. MVS systems using Version 5 and earlier releases of SAS require POINT= when using NOBS=. Finally, it is often good technique to create and process a summary dataset, rather than the original. The two PROC SUMMARYs above would work equally well on the original dataset VACATION. However, they would take longer to run.

Now we can use the macro variables &NROWS and &NCOLS to define variables and read in the data.

```

DATA ONEOBS
(DROP=DUMMY NAME PLACE TOTCOST);
ARRAY NAMES {&NROWS}
$ 12 NAME1-NAME&NROWS;
ARRAY PLACES {&NCOLS}
$ 12 PLACE1-PLACE&NCOLS;
ARRAY COSTS {*};
%DO ROW=1 %TO &NROWS;
    ROW&ROW.COL1-ROW&ROW.COL&NCOLS
%END;
;
DO DUMMY=1 TO &NROWS;
    SET NAMES;
    NAMES{DUMMY}=NAME;
END;
DO DUMMY=1 TO &NCOLS;
    SET PLACES;
    PLACES{DUMMY}=PLACE;
END;
DO DUMMY=1 TO &NROWS * &NCOLS;
    SET TOTALS (KEEP=TOTCOST);
    COSTS{DUMMY} = TOTCOST;
END;

```

The last DO loop makes a big assumption, namely that there is a COST for each NAME/PLACE combination. What if Bob never went to Arruba? It becomes tougher, but never impossible, to assign the TOTCOST variables to the proper elements of the COSTS array. Replace the last DO loop with:

```

DO UNTIL (EOF);
  SET TOTALS END=EOF;
  DO ROW=1 TO &NROWS;
    IF NAME=NAME{ROW}
      THEN GOTO COLS;
  END;
  COLS:
  DO COL=1 TO &NCOLS;
    IF PLACE=PLACE{COL}
      THEN GOTO ASSIGN;
  END;
  ASSIGN:
  DUMMY = (ROW-1)*&NCOLS + COL;
  COSTS{DUMMY} = TOTCOST;
END;

```

If COSTS were a two-dimensional array, the problem would become slightly easier.

Next, suppose we have to move in the opposite direction. The ONEOBS dataset exists, has a proper window definition to display it (that requires more macro programming which we will see later), and the DATA step is validating the elements of the COSTS array. If there is a problem with the Nth element, how can we retrieve the row and column to construct an error message such as:

PROBLEM WITH COST AT ROW 3 COLUMN 2

This message would be constructed by a statement such as:

```

_MSG_ = 'PROBLEM WITH COST AT ROW '
      || PUT(ROW,1.) || ' COLUMN '
      || PUT(COL,1.);

```

The program must compute values for ROW and COL based on the available incoming information: &NROWS and &NCOLS, plus the implied structure of the COSTS array. Given that N is the index variable for COSTS, the Nth element of COSTS falls into these rows and columns:

```

ROW = CEIL(N/&NCOLS);
COL = MOD(N,&NCOLS);
IF COL = 0 THEN COL = &NCOLS;

```

Since all NAMES and PLACES are variables as well, a small additional step would improve the message to something like:

PROBLEM WITH BOB'S ARRUBA TRIP

That final step is left as an exercise for the reader. The moral here is twofold: (1) keep the user well-informed, and (2) to accomplish step 1, deliver messages which the user will understand without straining.

Let's switch to another set of data and displays. Consider the master dataset (ITEMS) for a food coupon club. Each observation contains four variables:

```

ITEM          $ 12
  (Item for which coupon can be used)
DISCOUNT      8
  (Coupon value -- cents off)
STOCK           8
  (Number of coupons in stock)
HISTORY         8
  (Number of this type of coupon
   processed in the past year)

```

Each club member selects up to 20 coupons at a time and receives a maximum of 1 coupon per item. A sample screen appears at the top of the next page.

One programming objective is to make life as easy as possible for the user. Here, this means removing from the screen any ITEM which currently has no coupons available (STOCK=0). The basic approach will be that foods are not hard-coded on the window definition. Instead, MILK, COFFEE, etc. are variable values which are protected when displayed. The first DATA step selects and counts available ITEMS, collecting them into a single observation:

```

DATA WITHCOUP;
SET ITEMS END=EOF;
ARRAY ITEMS {45} $ 12 ITEM1-ITEM45;
RETAIN ITEM1-ITEM45;
IF STOCK > 0 THEN DO;
  TOTITEMS + 1;
  ITEMS{TOTITEMS} = ITEM;
END;
IF EOF THEN CALL
SYMPUT('TOTITEMS',TOTITEMS);

```

ITEMS

Command ---->

PUT AN "X" BEFORE EACH ITEM SELECTED THIS WEEK, UP TO A MAXIMUM OF 20 ITEMS.

- MILK	- CHERRIES	- GRAPEFRUIT
- COFFEE	- STRAWBERRIES	- STEAK
- TEA	- ORANGES	- PORK CHOPS
- SODA	- LETTUCE	- CHICKEN
- JUICE	- TOMATOES	- FISH
- EGGS	- CUCUMBERS	- SCALLOPS
- BACON	- MUSHROOMS	- STRINGBEANS
- BAGELS	- CARROTS	- POTATOES
- MUFFINS	- CELERY	- CORN
- CEREAL	- RADISHES	- YAMS
- SAUSAGES	- OLIVE OIL	- ARTICHOKE
- APPLES	- VINEGAR	- ONIONS
- BANANAS	- CHEESE	- TOFU
- PEACHES	- PARSLEY	- SPAGHETTI
- PEARS	- MINT	- RICE

RIC

ITEMS

Command ---->

ENTER ITEM # 8 ---->

(IF YOU HAVE < 20 ITEMS, JUST HIT THE "ENTER" KEY WHEN YOU ARE DONE.)

- 1 - APPLES
- 2 - BANANAS
- 3 - CHERRIES
- 4 - CHICKEN
- 5 - SCALLOPS
- 6 - ONIONS
- 7 - ARTICHOKE

RIC

The next DATA step uses the macro variable TOTITEMS to construct the WINDOW statement. This application assumes that the data will fit on the screen; there will never be more than 45 items to display.

```
DATA CHOICES;
SET WITHCOUP
  (KEEP=ITEM1-ITEM&TOTITEMS);
LENGTH X1-X&TOTITEMS $ 1;
WINDOW COUPONS
  #1 @5 'PUT AN "X" BEFORE EACH'
    @28 'ITEM SELECTED THIS WEEK,'
    @53 'UP TO A MAXIMUM'
    @69 'OF 20 ITEMS.'
  %DO DUMMY=1 %TO &TOTITEMS;
    %IF &DUMMY < 16 %THEN
      #(&DUMMY+3)
      @10 X&DUMMY $1.
      ATTR=UNDERLINE
      @12 ITEM&DUMMY $12.
      PROTECT=YES;
    %ELSE %IF &DUMMY < 31 %THEN
      #(&DUMMY-12)
      @30 X&DUMMY $1.
      ATTR=UNDERLINE
      @32 ITEM&DUMMY $12.
      PROTECT=YES;
    %ELSE #(&DUMMY-27)
      @50 X&DUMMY $1.
      ATTR=UNDERLINE
      @52 ITEM&DUMMY $12.
      PROTECT=YES;
  %END;
;
```

Although incomplete, this DATA step shows how macro language can limit processing to just the needed variables. Only selected variables (&TOTITEMS) were brought in, and the window will define a display containing only needed variables. If there were fewer than 45 items, the window would contain a blank section, but the cursor would never stop within that section. Macro language prevents that section from being defined within the window. And the user never calls up thinking that the window must have been broken.

Suppose the coupon club stocked thousands of different coupons, so that one screen could not display them all. One approach to the problem involves:

1. Have the user key in each desired ITEM.
2. Display a running record of all selected ITEMS.
3. Inform the user if the ITEM keyed in is not in stock or is not available through the club.

After the user has entered 7 items, the window might look like the bottom of the previous page.

Macro language can develop formats which let the user know when an ITEM is unavailable or out of stock. For example:

```
DATA _NULL_;
SET ITEMS (KEEP=ITEM STOCK) END=EOF;
CALL SYMPUT('VAL' || _N_, ITEM);
IF STOCK > 0 THEN
  CALL SYMPUT('OUT' || _N_, ' ');
ELSE CALL SYMPUT('OUT' || _N_,
  'TEMPORARILY OUT OF STOCK');
IF EOF THEN CALL
SYMPUT('TOTITEMS', _N_);
RUN;

PROC FORMAT;
VALUE $INSTOCK
  %DO DUMMY=1 %TO &TOTITEMS;
    "VAL&DUMMY" = "OUT&DUMMY"
  %END;
  OTHER = "NOT AN AVAILABLE ITEM";
```

When in final form, the macro programming statements would not stand alone in open SAS code. They would be incorporated into the definition of one or more SAS macros.

Once this format exists, the DATA step can use it to respond to user input. See the complete program on the next page. The user receives appropriate messages whenever he has entered an ITEM which is temporarily out of stock or which is not carried by the club.

A user-friendly system could analyze this data in many ways. Consider a simple regression. With only three numeric variables to work with, it is easy to write the program:

```
PROC REG DATA=ITEMS;
MODEL HISTORY = DISCOUNT STOCK;
```

Sample program which responds to user input using a format:

```
DATA TEMP (KEEP=ITEM1-ITEM20);
LENGTH ITEM1-ITEM20 $ 12;
ARRAY ITEMS {20} ITEM1-ITEM20;
N=1;
WINDOW ITEMS
  #1 @1 'ENTER ITEM #' @13 N 2. PROTECT=YES @16 '==>'
    @21 ITEM $12. ATTR=UNDERLINE
  #3 @1 '(IF YOU WANT < 20 ITEMS, JUST'
    @31 'HIT THE "ENTER" KEY WHEN YOU ARE DONE.)'
  #6 @10 ITEM1 $CHAR17. PROTECT=YES
    @40 ITEM11 $CHAR17. PROTECT=YES
  #7 @10 ITEM2 $CHAR17. PROTECT=YES
    @40 ITEM12 $CHAR17. PROTECT=YES
  #8 @10 ITEM3 $CHAR17. PROTECT=YES
    @40 ITEM13 $CHAR17. PROTECT=YES
  #9 @10 ITEM4 $CHAR17. PROTECT=YES
    @40 ITEM14 $CHAR17. PROTECT=YES
  #10 @10 ITEM5 $CHAR17. PROTECT=YES
    @40 ITEM15 $CHAR17. PROTECT=YES
  #11 @10 ITEM6 $CHAR17. PROTECT=YES
    @40 ITEM16 $CHAR17. PROTECT=YES
  #12 @10 ITEM7 $CHAR17. PROTECT=YES
    @40 ITEM17 $CHAR17. PROTECT=YES
  #13 @10 ITEM8 $CHAR17. PROTECT=YES
    @40 ITEM18 $CHAR17. PROTECT=YES
  #14 @10 ITEM9 $CHAR17. PROTECT=YES
    @40 ITEM19 $CHAR17. PROTECT=YES
  #15 @10 ITEM10 $CHAR17. PROTECT=YES
    @40 ITEM20 $CHAR17. PROTECT=YES;
WINDOW FINISHED ROWS=4
  #1 @1 'ALL ITEMS COMPLETE. PLEASE HIT THE "ENTER" KEY.';
LENGTH MESSAGE $24;
DO UNTIL (N=21);
SHOWTIME: DISPLAY ITEMS BLANK;
  MESSAGE = PUT(ITEM,$INSTOCK.);
  IF ITEM > ' ' THEN DO;
    IF MESSAGE=' ' THEN DO;
      ITEMS{N}=PUT(N,2.) || ' - ' || ITEM;
      ITEM=' ';
      N+1;
    END;
    ELSE GOTO SHOWTIME;
  END;
  ELSE N=21;
END;
DISPLAY FINISHED BLANK;
OUTPUT;
STOP;
RUN;
```

However, suppose there were hundreds of variables, both character and numeric, but the objective were similar: select an independent variable, and run a regression against all other numeric variables in the dataset. The generalized form would be:

```
PROC REG DATA=ITEMS;
MODEL selected numeric = all other numerics;
```

The following macro code generates such a program, subject to two restrictions: (1) the incoming dataset must be ALL.ITEMS, and (2) the dependent variable name must exist and be numeric. (Appropriate enhancements would eliminate such restrictions, but that is not the point here.)

```
%MACRO RUNREG (DEP=,INDEP=);
  PROC CONTENTS DATA=ITEMS NOPRINT
    OUT=ITEMS (KEEP=NAME TYPE);

  DATA _NULL_;
  SET ITEMS;
  IF TYPE=1 AND NAME ^= "&DEP"
  THEN CALL SYMPUT
    ('INDEP',"&INDEP " || TRIM(NAME));

  PROC REG DATA=ITEMS;
  MODEL &DEP = &INDEP;
  RUN;

%MEND RUNREG;

%RUNREG (DEP=HISTORY)
```

PROC CONTENTS creates an output dataset (ITEMS) containing one observation for each variable in the input dataset. NAME (\$ 8) is the variable name, and TYPE (numeric) is 1 for numeric variables and 2 for character variables. The program constructs &INDEP by examining each variable from ALL.ITEMS, checking whether it is character or numeric, and checking whether or not it is the dependent variable.

This combination of Macro language with PROC CONTENTS output solves many types of problems. Another example attempts to use the SORTSIZE option on PROC SORT. To best utilize memory and CPU time, SORTSIZE should be $((\text{VARLENGTHS} + 10) * 2) + 100$, where VARLENGTHS is the sum of the lengths of all variables in the dataset being sorted.

```
PROC CONTENTS DATA=ITEMS NOPRINT
  OUT=LENGTHS (KEEP=LENGTH);

DATA _NULL_;
SET LENGTHS END=EOF;
VARLEN + LENGTH;
IF EOF THEN DO;
  SIZE = ((VARLEN+10) * 2) + 100;
  CALL SYMPUT('SIZE',SIZE);
END;

PROC SORT DATA=ITEMS
  SORTSIZE=&SIZE;
BY HISTORY;
RUN;
```

A similar macro could check that CLASS variables are character, since numeric CLASS variables can generate tons of output.

The last topic we will examine is naming screens and datasets. SAS software frequently displays these names, so it is important that you choose meaningful names. Consider this sort of message from the user's point of view:

```
Edit SAS dataset:  WORK.X
Edit SAS dataset:  LIB.TEMP
```

These messages only serve to confuse. Since you cannot suppress these displays, at least choose names which describe the data being displayed:

```
Edit SAS dataset:  SUMMER.VACATION
Edit SAS dataset:  FAVORITE.HOBBY
```

It is not necessary to store every SAS dataset in a separate library to accomplish this. Remember, the same library can be assigned different librefs by using many LIBNAME statements.

The same principle applies to displaying screen names as opposed to dataset names. Select meaningful screen names, if the user will end up seeing the name. In fact, modular systems which use the same screen to display more than one dataset may apply this principle by giving the same screen different names. Once the

screen definition is cast in concrete, the screen may be stored several times under different names. Alternatively, DATA steps allow the screen name to be a macro variable. For example, consider:

```
%MACRO HOTEL (PLACE=);
  DATA &PLACE;
  SET &PLACE;
  WINDOW &PLACE
    window definition
  ;
  DO UNTIL (UPCASE(_CMD_)='DONE');
    DISPLAY &PLACE;
  END;
%MEND HOTEL;
%HOTEL (PLACE=HILTON)
```

This macro uses the same window to edit various datasets. However, the window name which appears in the upper left hand corner of the screen is determined when the macro is invoked.

Techniques will change, but the objective remains the same. Focus on providing the user with clear displays, minimal work, and easy-to-understand instructions. Programming techniques will change as the SAS software incorporates more and more features. For example, version 5 %PUT statements truncated the message when the length of the string being written exceeded the display line length. In order to write the value of a lengthy macro variable, the programmer had to write a macro that would break up the value and write it over many lines. Under version 6, %PUT handles that automatically. As the software is enhanced, I hope that these features will be incorporated into screen design products:

1. Cursor control as well as message control.
2. An option to eliminate the Command Line ==> section of screens.
3. Execution-time control over screen attributes. One example: At display time, PROTECT a field which was unprotected when the screen was designed.

Finally, this paper contains a sample of the tools and techniques -- as much as I could reasonably fit. If you have questions or comments about these or other techniques, feel free to call (or write):

Bob Virgile
Robert Virgile Associates, Inc.
3 Rock Street
Woburn, MA 01801

*SAS is a registered trademark of SAS Institute, Inc., Cary, NC.